C für Mikrocontroller

Einführung / Überblick

Ein Volksmärchen:
"C ist eine maschinenunabhängige höhere Programmiersprache. Ein richtig geschriebenes C-Programm läuft auf jeder Maschine. Man muß es nur entsprechend übersetzen." Schön wär's ...

Die harten Tasachen:
C ist NICHT maschinenunabhängig – vor allem nicht im Bereich der Mikrocontroller. Zudem ist es nicht einmal unabhängig vom Compiler. Der Compiler macht, was er will, und nimmt keine Rücksicht auf subtile Programmierabsichten, wie sie für die Mikrocontrollerprogrammierung typisch sind. Das betrifft u. a.

- die Unterbringung von Variablen in verschiedenartigen Speicherbereichen (SRAM, Flash, EEPROM usw.),
- die Notwendigkeit, auf manche Variable immer wieder zuzugreifen, obwohl sie im Programmverlauf eigentlich nicht verändert wurden (weil es sich um Werte handelt, die von außen oder anderswo her kommen (Eingaben)),
- Befehlsfolgen, die eigentlich nichts berechnen, sondern nur Zeit verbrauchen.

Da solche Unterscheidungen nicht in den Sprachstandard eingebaut sind, hat sie jeder Compilerautor gleichsam auf eigene Faust definiert. Selbst bei Beschränkung auf eine einzige Zielarchitektur (z. B. Atmel AVR) entspricht somit jeder Compiler praktisch einem eigenen C-Dialekt:

- beim Übergang von Compiler A auf Compiler B muß im Quelltext geändert werden.
- wer die Eigentümlichkeiten der jeweiligen C-Umgebung nicht kennt, kann entsprechend geschriebene Quelltexte nicht verstehen.

– C, ohnehin schon häßlich genug, wird noch häßlicher. –

C als Mittel zur Gewährleistung der Maschinenunabhängigkeit
Nur sehr beschränkt. In der Theorie schaut alles viel großartiger aus, als es wirklich ist. Wenn man Maschinenunabhängigkeit haben will, muß man entsprechend entwickeln. Diese Entwicklungsmethodik kostet aber Zeit.

C als Mittel zur Arbeitserleichterung
Durchaus gegeben. Heutzutage das Hauptmotiv, weshalb man C einsetzt. Man darf sich nur nicht der Illusion hingeben, die Entwicklungsleistung wären maschinenunabhängig und durch einfaches Kompilieren auf x-beliebige andere Maschinen zu übertragen. Deshalb ist es gängige Praxis, bei einer Prozessorachitektur und einer Programmentwicklungsumgebung zu bleiben – und deren Nachteile ggf. hinzunehmen (zumal professionelle Entwicklungsumgebungen auch richtig Geld kosten – die Annahme, es sei alles umsonst, ist eine irrtümliche ...).

Datentypen

Typumwandlung

Speicheraufteilung / Pointer

Speichermodelle

Standardgemäße Ein- und Ausgabe

Maschinenspezifische Register

Elementare Ein- und Ausgabe

Interruptbehandlung

Bitoperationen

Schnittstellen zur Assemblerprogrammierung

Compiler-Direktiven

Abweichungen vom ANSI-Standard

Weshalb nehmen wir eigentlich C? (Es zählen nur vernünftige Gründe.)

- um schneller voranzukommen.
- um tatsächlich – soweit es irgendwie geht – trotz allem maschinenunabhängig zu werden (Portabilität).

Der Compiler kümmert sich um

- den Unterprogrammaufruf,
- das Belegen, Retten und Wiedereinstellen von Registern,
- die Speicherbelegung,
- das Adressieren der Variablen,
- überschaubare Kontrollstrukturen (Entscheidungen, Schleifen usw.),
- die elementare Arithmetik (= alle vier Grundrechenarten mit den gängigen Datentypen),
- weitere elementare Funktionen (Bibliotheksroutinen).

Andernfalls – wenn wir von Grund auf alles in Assembler programmieren – müßten wir all dies zu Fuß tun bzw. eigene Lösungen finden.

Gutgemeinte Empfehlungen

Teile und herrsche – das Problem aufteilen: was ist sozusagen innere Programmlogik (maschinenunabhängig), was ist echte Ein- und Ausgabe (unvermeidlich maschinenspezifisch), was sind organisatorische Kleinigkeiten (z. B. Speicheraufteilung)?

Von oben herangehen (Top-Down) und schrittweise verfeinern. Problemlösung zunächst – ganz abstrakt – nur in C bearbeiten: Variablen – Programmlogik – Funktionen an den Schnittstellen zur Hardware ("logische" Ein- und Ausgabe).

Die typische C-Schnittstelle ist ein Funktionsaufruf. Alles,was sich nicht mit standardgemäßem (ANSI-) C ausdrücken läßt, zunächst mit Funktionen erledigen (und wenn diese zunächst bloße Attrappen (Dummies) sind).

Diese Funktionen schrittweise verfeinern. Wenn erforderlich, durch Assembler-Programmstücke ersetzen. (Notfalls durch Suchen und Ersetzen im Editor.)

Wenn möglich, Programmlogik auf PC ausprobieren. Funktionskörper ggf. passend abwandeln. Z. B. Eingabe über Tastatur oder Schnittstelle, Ausgabe auf Bildschirm bis hin zur Nachbildung der Peripherie mit Windows-Steuerelementen oder Kombination aus PC und angeschlossener Peripherie-Nachbildung.

Mit C assemblermäßig programmieren. Möglichkeiten der Sprache nicht bis zum äußersten ausnutzen.

Man hält sich entweder an akademische Weisheiten oder schreibt Programme, die in endlicher Zeit fertig werden und vernünftig laufen. Keine Angst vor GOTOs, globalen Variablen usw. Der wirkliche Könner ist nicht derjenige, der solche Programmiertechniken ängstlich vermeidet, sondern der weiß, wo man sie einsetzt und wo nicht.

Compilerspezifische Komfort-Funktionen nicht ausnutzen.

Systematisch entwickeln.

Alles richtig dokumentieren.

C-Quelltext ist keine Programmdokumentation!

Das Codieren sollte höchstens 30% der Programmentwicklungzeit erfordern.

Erst denken, dann hacken.